Algorithmen und Datenstrukturen

A4. Sortieren II: Mergesort

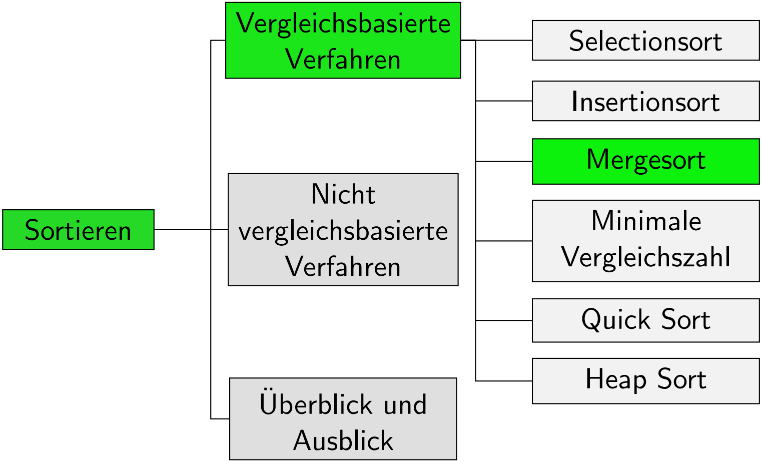
Marcel Lüthi and Gabriele Röger

Universität Basel

21. Februar 2019

Mergesort

Sortierverfahren



Mergesort: Idee

- Beobachtung: zwei bereits sortierte Sequenzen lassen sich leicht zu einer sortierten Sequenz vereinen.
- Sequenzen mit einem oder keinem Element sind sortiert.
- Idee für längere Sequenzen:
  - Teile Eingabesequenz in zwei etwa gleich grosse Teilbereiche
  - Rekursiver Aufruf für beide Teilmengen
  - Füge nun sortierte Teilbereiche zusammen.
- Teile-und-Herrsche-Ansatz (divide and conquer)

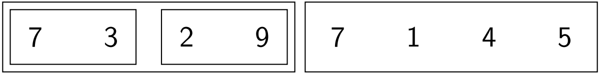
Mergesort: Illustration

7	3	2	9	7	1	4	5

Mergesort: Illustration

7	3	2	9	7	1	4	5

Mergesort: Illustration



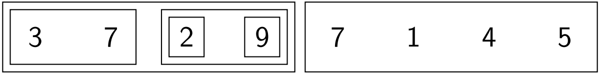
Mergesort: Illustration

7	3	2	9	7	1	4	5

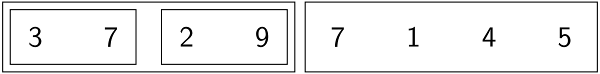
Mergesort: Illustration

3	7	2	9	7	1	4	5

Mergesort: Illustration



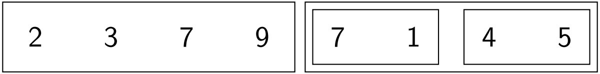
Mergesort: Illustration



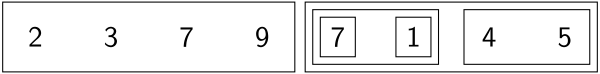
Mergesort: Illustration

2 3 7 9	7 1 4 5

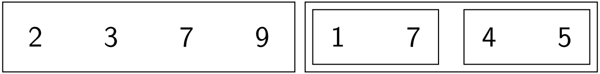
Mergesort: Illustration



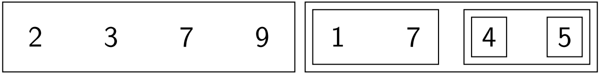
Mergesort: Illustration



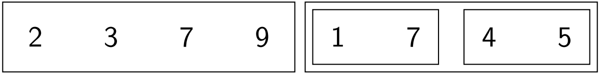
Mergesort: Illustration



Mergesort: Illustration



Mergesort: Illustration



Mergesort: Illustration

2	3	7	9	1	4	5	7

Mergesort: Illustration

1	2	3	4	5	7	7	9

Merge-Schritt

Verbinden der Teillösungen

- Indizes lo $\leq$ mid $<$ hi
- Annahme: array[lo] bis array[mid] und array[mid+1] bis array[hi] sind bereits sortiert
- Ziel: array[lo] bis array[hi] ist sortiert
- Idee: gehe parallel von vorne nach hinten durch beide Teilbereiche und sammle das jeweils kleinere Element auf
- Verwendet zusätzlichen Speicher für aufgesammelte Werte

Verbinden der Teillösungen: Beispiel

Array tmp hat gleiche Grösse wie Eingabearray.
Initialisierung: i := lo, j := mid + 1, k := lo

Verbinden der Teillösungen: Beispiel

Array tmp hat gleiche Grösse wie Eingabearray.
Initialisierung: i := lo, j := mid + 1, k := lo

a

	lo,i		mid	j	hi	
···	2	4	5	4	7	···

tmp

	k					
···						···

a

	lo	i	mid	j	hi	
···	2	4	5	4	7	···

tmp

		k				
···	2					···

$a[i] \leq a[j] \Rightarrow tmp[k] = a[i]$

Verbinden der Teillösungen: Beispiel

Array tmp hat gleiche Grösse wie Eingabearray.
Initialisierung: i := lo, j := mid + 1, k := lo

			a							tmp				
	lo,i		mid	j	hi			k						
···	2	4	5	4	7	···	···						···	
	lo	i	mid	j	hi				k					a[i]≤a[j] ⇒ tmp[k] = a[i]
···	2	4	5	4	7	···	···	2					···	
	lo		mid,i	j	hi					k				a[i]≤a[j] ⇒ tmp[k] = a[i]
···	2	4	5	4	7	···	···	2	4				···	

Verbinden der Teillösungen: Beispiel

Array tmp hat gleiche Grösse wie Eingabearray.
Initialisierung: i := lo, j := mid + 1, k := lo

	a								tmp					
	lo,i		mid	j	hi			k						
…	2	4	5	4	7	…	…						…	
	lo	i	mid	j	hi				k					$a[i] \leq a[j] \Rightarrow tmp[k] = a[i]$
…	2	4	5	4	7	…	…	2					…	
	lo		mid,i	j	hi					k				$a[i] \leq a[j] \Rightarrow tmp[k] = a[i]$
…	2	4	5	4	7	…	…	2	4				…	
	lo		mid,i		hi,j						k			$a[j] < a[i] \Rightarrow tmp[k] = a[j]$
…	2	4	5	4	7	…	…	2	4	4			…	

Verbinden der Teillösungen: Beispiel

Array tmp hat gleiche Grösse wie Eingabearray.
Initialisierung: $i := lo$, $j := mid + 1$, $k := lo$

	a		tmp	
lo,i · mid · j · hi	… 2 4 5 4 7 …	k (Pos. 1)	… ☐ ☐ ☐ ☐ ☐ …	
lo · i · mid · j · hi	… 2 4 5 4 7 …	k (Pos. 2)	… 2 ☐ ☐ ☐ ☐ …	$a[i] \leq a[j] \Rightarrow tmp[k] = a[i]$
lo · mid,i · j · hi	… 2 4 5 4 7 …	k (Pos. 3)	… 2 4 ☐ ☐ ☐ …	$a[i] \leq a[j] \Rightarrow tmp[k] = a[i]$
lo · mid,i · hi,j	… 2 4 5 4 7 …	k (Pos. 4)	… 2 4 4 ☐ ☐ …	$a[j] < a[i] \Rightarrow tmp[k] = a[j]$
lo · mid · i · hi,j	… 2 4 5 4 7 …	k (Pos. 5)	… 2 4 4 5 ☐ …	$a[i] \leq a[j] \Rightarrow tmp[k] = a[i]$

Verbinden der Teillösungen: Beispiel

Array tmp hat gleiche Grösse wie Eingabearray.
Initialisierung: i := lo, j := mid + 1, k := lo

	a		tmp	
lo,i · · mid · j · hi	⋯ [2][4][5][4][7] ⋯	k (Pos. 1)	⋯ [ ][ ][ ][ ][ ] ⋯	
lo · i · mid · j · hi	⋯ [2][4][5][4][7] ⋯	k (Pos. 2)	⋯ [2][ ][ ][ ][ ] ⋯	a[i]≤a[j] ⇒ tmp[k] = a[i]
lo · · mid,i · j · hi	⋯ [2][4][5][4][7] ⋯	k (Pos. 3)	⋯ [2][4][ ][ ][ ] ⋯	a[i]≤a[j] ⇒ tmp[k] = a[i]
lo · · mid,i · · hi,j	⋯ [2][4][5][4][7] ⋯	k (Pos. 4)	⋯ [2][4][4][ ][ ] ⋯	a[j]<a[i] ⇒ tmp[k] = a[j]
lo · · mid · i · hi,j	⋯ [2][4][5][4][7] ⋯	k (Pos. 5)	⋯ [2][4][4][5][ ] ⋯	a[i]≤a[j] ⇒ tmp[k] = a[i]
lo · · mid · i · hi	⋯ [2][4][5][4][7] ⋯		⋯ [2][4][4][5][7] ⋯	i>mid ⇒ tmp[k] = a[j]

Verbinden der Teillösungen: Algorithmus

```
def merge(array, tmp, lo, mid, hi):
    i = lo
    j = mid + 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        if j > hi or (i <= mid and array[i] <= array[j]):
            tmp[k] = array[i]
            i += 1
        else:
            tmp[k] = array[j]
            j += 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        array[k] = tmp[k]
```

Verbinden der Teillösungen: Algorithmus

```python
def merge(array, tmp, lo, mid, hi):
    i = lo
    j = mid + 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        if j > hi or (i <= mid and array[i] <= array[j]):
            tmp[k] = array[i]
            i += 1
        else:
            tmp[k] = array[j]
            j += 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        array[k] = tmp[k]
```

Auch korrekt für lo = mid = hi

Jupyter-Notebook

Jupyter-Notebook: `merge_sort.ipynb`

Top-Down-Mergesort

Mergesort: Algorithmus

rekursive Top-Down-Version

```
def sort(array):
    tmp = [0] * len(array)  # [0,...,0] with same size as array
    sort_aux(array, tmp, 0, len(array) - 1)

def sort_aux(array, tmp, lo, hi):
    if hi <= lo:
        return
    mid = lo + (hi - lo) // 2
    # //: Division mit Abrunden
    sort_aux(array, tmp, lo, mid)
    sort_aux(array, tmp, mid + 1, hi)
    merge(array, tmp, lo, mid, hi)
```

Mögliche Verbesserungen

- Auf kurzen Sequenzen ist Insertionsort schneller als Mergesort
  $\rightarrow$ verwende Insertionsort wenn hi - lo klein

Mögliche Verbesserungen

- Auf kurzen Sequenzen ist Insertionsort schneller als Mergesort
  → verwende Insertionsort wenn hi - lo klein
- Breche Merge-Schritt direkt ab, falls Positionen lo bis hi bereits vollständig sortiert

```
if array[mid] <= array[mid + 1]:
    return
```

Mögliche Verbesserungen

- Auf kurzen Sequenzen ist Insertionsort schneller als Mergesort
  → verwende Insertionsort wenn hi - lo klein
- Breche Merge-Schritt direkt ab, falls Positionen lo bis hi bereits vollständig sortiert

```
if array[mid] <= array[mid + 1]:
    return
```

- Kopieren von `tmp`-Ergebnis in merge kostet Zeit
  → tausche Rolle von `array` und `tmp`
  bei jedem rekursiven Aufruf

Merge-Schritt: Korrektheit

- Invariante: Am Ende jeder Schleifeniteration ist
  - `tmp[k]` $\leq$ `array[m]` für alle $i \leq m \leq$ mid, und
  - `tmp[k]` $\leq$ `array[n]` für alle $j \leq n \leq$ hi.
- `tmp` wird von vorne nach hinten beschrieben.
- Nach letzter Schleifeniteration gilt für alle lo $\leq r < s \leq$ hi, dass `tmp[`r`]`$\leq$`tmp[`s`]` (= Bereich ist sortiert).

Mergesort: Korrektheit

`sort_aux`:

- Induktionsbeweis über Bereichslänge $\mathsf{hi} - \mathsf{lo}$
- Basis $\mathsf{hi} - \mathsf{lo} = -1$: leerer Bereich ist sortiert.
- Basis $\mathsf{hi} - \mathsf{lo} = 0$: Bereich mit nur einem Element ist sortiert.

Mergesort: Korrektheit

`sort_aux`:

- Induktionsbeweis über Bereichslänge $\mathsf{hi} - \mathsf{lo}$
- Basis $\mathsf{hi} - \mathsf{lo} = -1$: leerer Bereich ist sortiert.
- Basis $\mathsf{hi} - \mathsf{lo} = 0$: Bereich mit nur einem Element ist sortiert.
- Induktionshypothese: Mergesort ist korrekt für alle $\mathsf{hi} - \mathsf{lo} < m$
- Induktionsschritt $(m - 1 \to m)$:

Mergesort: Korrektheit

`sort_aux`:

- Induktionsbeweis über Bereichslänge hi − lo
- Basis $\text{hi} - \text{lo} = -1$: leerer Bereich ist sortiert.
- Basis $\text{hi} - \text{lo} = 0$: Bereich mit nur einem Element ist sortiert.
- Induktionshypothese: Mergesort ist korrekt für alle $\text{hi} - \text{lo} < m$
- Induktionsschritt $(m - 1 \to m)$:
  Mergesort macht zwei rekursive Aufrufe mit
  $\text{hi} - \text{lo} \leq m/2 + 1$, danach ist die Eingabe jeweils zwischen lo und mid und zwischen mid + 1 und hi sortiert (lt. Ind.-hyp).

Mergesort: Korrektheit

`sort_aux`:

- Induktionsbeweis über Bereichslänge $\text{hi} - \text{lo}$
- Basis $\text{hi} - \text{lo} = -1$: leerer Bereich ist sortiert.
- Basis $\text{hi} - \text{lo} = 0$: Bereich mit nur einem Element ist sortiert.
- Induktionshypothese: Mergesort ist korrekt für alle $\text{hi} - \text{lo} < m$
- Induktionsschritt $(m - 1 \to m)$:
  Mergesort macht zwei rekursive Aufrufe mit $\text{hi} - \text{lo} \leq m/2 + 1$, danach ist die Eingabe jeweils zwischen lo und mid und zwischen $\text{mid} + 1$ und hi sortiert (lt. Ind.-hyp).

  Wir wissen bereits, dass der Merge-Schritt korrekt ist, also ist am Ende der gesamte Bereich zwischen lo und hi sortiert.

Mergesort: Korrektheit

`sort_aux`:

- Induktionsbeweis über Bereichslänge $\mathrm{hi} - \mathrm{lo}$
- Basis $\mathrm{hi} - \mathrm{lo} = -1$: leerer Bereich ist sortiert.
- Basis $\mathrm{hi} - \mathrm{lo} = 0$: Bereich mit nur einem Element ist sortiert.
- Induktionshypothese: Mergesort ist korrekt für alle $\mathrm{hi} - \mathrm{lo} < m$
- Induktionsschritt ($m - 1 \to m$):
  Mergesort macht zwei rekursive Aufrufe mit $\mathrm{hi} - \mathrm{lo} \leq m/2 + 1$, danach ist die Eingabe jeweils zwischen lo und mid und zwischen mid + 1 und hi sortiert (lt. Ind.-hyp).

  Wir wissen bereits, dass der Merge-Schritt korrekt ist, also ist am Ende der gesamte Bereich zwischen lo und hi sortiert.

Mergesort: Ruft `sort_aux` für gesamten Bereich auf,
daher ist am Ende die gesamte Eingabe sortiert.

Mergesort: Eigenschaften

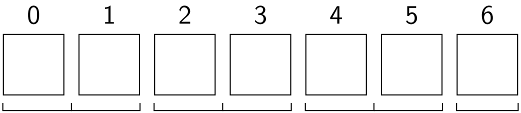
- nicht in-place: verwendet zusätzlichen Speicherplatz für `tmp` und für Aufrufstapel (call stack)
- Zeitbedarf: nicht adaptiv (ausser mit Mergeabbruch-Verbesserung)
  genauere Analyse: nächste Woche
- stabil: `merge` präferiert `array[i]`,
  wenn `array[i]` gleich `array[j]`.

Bottom-Up-Mergesort

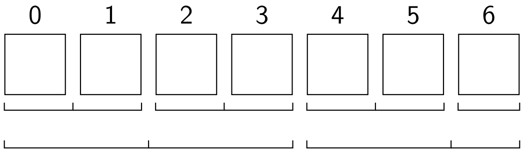
Bottom-Up-Version

0	1	2	3	4	5	6

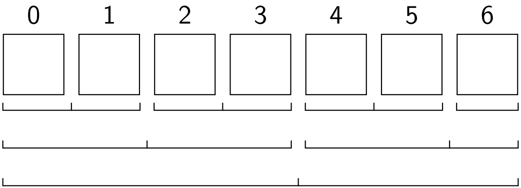
Bottom-Up-Version



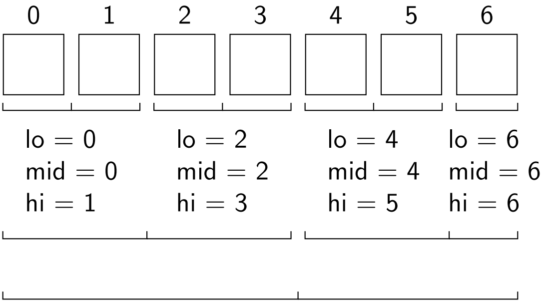
Bottom-Up-Version



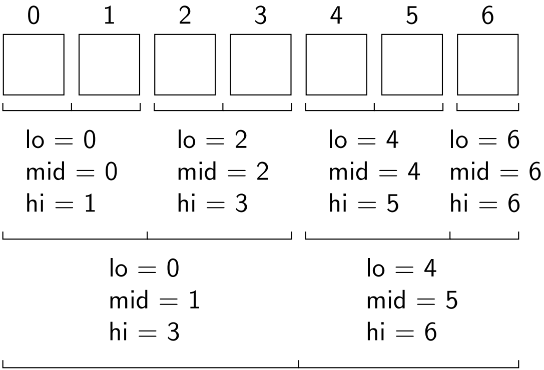
Bottom-Up-Version



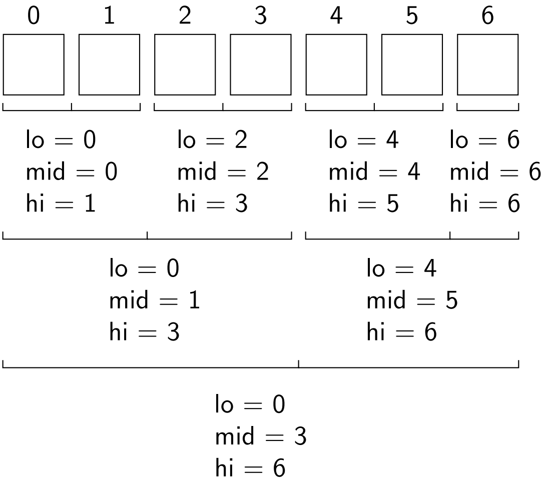
Bottom-Up-Version



Bottom-Up-Version



Bottom-Up-Version



Bottom-Up-Mergesort: Algorithm

iterative Bottom-Up-Version

```
def sort(array):
    n = len(array)
    tmp = [0] * n
    length = 1
    while length < n:
        lo = 0
        while lo < n - length:
            mid = lo + length - 1
            hi = min(lo + 2 * length - 1, n - 1)
            merge(array, tmp, lo, mid, hi)
            lo += 2 * length
        length *= 2
```

Zusammenfassung

Zusammenfassung

- Mergesort ist ein Teile-und-Herrsche-Verfahren, das den zu sortierenden Bereich in zwei etwa gleich grosse Bereiche teilt.
- Der Merge-Schritt führt zwei bereits sortierte Teilbereiche zusammen.
- Mergesort ist stabil, arbeitet aber nicht in-place.
- Die Top-Down-Variante ist ein rekursives Verfahren.
- Die Bottom-Up-Variante ist ein iteratives Verfahren.